

The libMesh Finite Element Library

a case for Object-Oriented High-Performance Computing

Benjamin S. Kirk[†]

`benjamin.kirk@nasa.gov`

John W. Peterson[‡]

`peterson@cfdlab.ae.utexas.edu`

Roy H. Stogner^{*}

`roystgnr@ices.utexas.edu`

[†]NASA Lyndon B. Johnson Space Center

[‡]Idaho National Labs

^{*}The University of Texas at Austin

June 17, 2013



- 1 Introduction
 - Background
 - The `libMesh` Software Library
 - Software Reusability
 - Library Design
- 2 Motivation
 - Application Results
- 3 Approach to Software Development
- 4 A Generic Boundary Value Problem
- 5 Key Data Structures
 - The Mesh Class
 - The EquationSystems Class
- 6 Weighted Residuals
- 7 Poisson Equation
- 8 Other Examples
- 9 Some Extensions



1 Introduction

- Background
- The `libMesh` Software Library
- Software Reusability
- Library Design

2 Motivation

- Application Results

3 Approach to Software Development

4 A Generic Boundary Value Problem

5 Key Data Structures

- The Mesh Class
- The EquationSystems Class

6 Weighted Residuals

7 Poisson Equation

8 Other Examples

9 Some Extensions



Background

- Modern simulation software is **complex**:
 - Implicit numerical methods
 - Massively parallel computers
 - Adaptive methods
 - Multiple, coupled physical processes
- There are a host of existing software libraries that excel at treating various aspects of this complexity.
- Leveraging existing software whenever possible is the most efficient way to manage this complexity.



Background

- Modern simulation software is **multidisciplinary**:
 - Physical Sciences
 - Engineering
 - Computer Science
 - Applied Mathematics
 - ...
- It is not reasonable to expect a single person to have all the necessary skills for developing & implementing high-performance numerical algorithms on modern computing architectures.
- Teaming is a prerequisite for success.



Background

- A large class of problems are amenable to **mesh based** simulation techniques.
- Consider some of the major components such a simulation:



Background

- A large class of problems are amenable to **mesh based** simulation techniques.
- Consider some of the major components such a simulation:
 - 1 Read the mesh from file
 - 2 Initialize data structures
 - 3 Construct a discrete representation of the governing equations
 - 4 Solve the discrete system
 - 5 Write out results
 - 6 Optionally estimate error, refine the mesh, and repeat



Background

- A large class of problems are amenable to **mesh based** simulation techniques.
- Consider some of the major components such a simulation:
 - 1 Read the mesh from file
 - 2 Initialize data structures
 - 3 Construct a discrete representation of the governing equations
 - 4 Solve the discrete system
 - 5 Write out results
 - 6 Optionally estimate error, refine the mesh, and repeat
- With the exception of step 3, the rest is *independent* of the class of problems being solved.



Background

- A large class of problems are amenable to **mesh based** simulation techniques.
- Consider some of the major components such a simulation:
 - 1 Read the mesh from file
 - 2 Initialize data structures
 - 3 Construct a discrete representation of the governing equations
 - 4 Solve the discrete system
 - 5 Write out results
 - 6 Optionally estimate error, refine the mesh, and repeat
- With the exception of step 3, the rest is *independent* of the class of problems being solved.
- This allows the major components of such a simulation to be abstracted & implemented in a reusable software library.



The libMesh Software Library

- In 2002, the libMesh library began with these ideas in mind.
- Primary goal is to provide data structures and algorithms that can be shared by disparate physical applications, that may need some combination of
 - Implicit numerical methods
 - Adaptive mesh refinement techniques
 - Parallel computing
- Unifying theme: mesh-based simulation of partial differential equations (PDEs).



The libMesh Software Library

Key Point

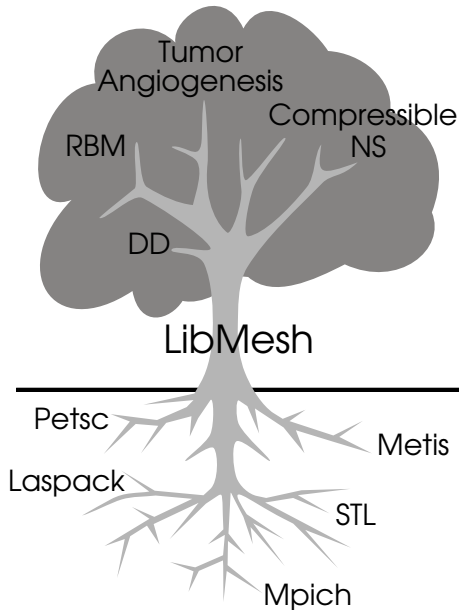
- The libMesh library is designed to be used by students, researchers, scientists, and engineers as a tool for **developing simulation codes** or as a tool for **rapidly implementing a numerical method**.
- libMesh is not an application code.
- It does not “solve problem XYZ.”
 - It can be used to help you develop an application to solve problem XYZ, and to do so quickly with advanced numerical algorithms on high-performance computing platforms.



Software Reusability

- At the inception of `libMesh` in 2002, there were many high-quality software libraries that implemented some aspect of the end-to-end PDE simulation process:
 - Parallel linear algebra
 - Partitioning algorithms for domain decomposition
 - Visualization formats
 - ...
- A design goal of `libMesh` has always been to provide flexible & extensible interfaces to existing software whenever possible.
- We implement the “glue” to these pieces, as well as what we viewed as the missing infrastructure:
 - Flexible data structures for the discretization of spatial domains and systems of PDEs posed on these domains.





Library Structure

- Basic libraries are libMesh's "roots"
- Application "branches" built off the library "trunk"



The “Glue”

- The C++ programming language provides a powerful abstraction mechanism for separating a software interface from its implementation.
- The notion of **Base Classes** defining an abstract interface and **Derived Classes** implementing the interface is key to this programming model.



The “Glue”

- The C++ programming language provides a powerful abstraction mechanism for separating a software interface from its implementation.
- The notion of **Base Classes** defining an abstract interface and **Derived Classes** implementing the interface is key to this programming model.
- The classic C++ example: Shapes.

```
int main ()
{
    /* using shapes polymorphically */
    Shape * shapes[2];
    shapes[0] = new Rectangle (10, 20, 5, 6);
    shapes[1] = new Circle (15, 25, 8);

    for (int i=0; i<2; ++i)
        shapes[i]->Draw();

    ...
}
```



Abstract Shape

```
/* abstract interface declaration */  
class Shape  
{  
public:  
    virtual void Draw () = 0;  
    virtual void MoveTo (int newX, int newY) = 0;  
    virtual void RMoveTo (int dx, int dy) = 0;  
};
```



Specific Shape: Rectangle

```
/* Class Rectangle */  
class Rectangle : public Shape  
{  
public:  
    Rectangle (int x, int y, int w, int h);  
    virtual void Draw ();  
    virtual void MoveTo (int newX, int newY);  
    virtual void RMoveTo (int dx, int dy);  
    virtual void SetWidth (int newWidth);  
    virtual void SetHeight (int newHeight);  
  
private:  
    int x, y;  
    int width;  
    int height;  
};
```



Specific Shape: Circle

```
/* Class Circle */  
class Circle : public Shape  
{  
public:  
    Circle (int initx, int inity, int initr);  
    virtual void Draw ();  
    virtual void MoveTo (int newx, int newy);  
    virtual void RMoveTo (int dx, int dy);  
    virtual void SetRadius (int newRadius);  
  
private:  
    int x, y;  
    int radius;  
};
```



Object Polymorphism

```
int main ()
{
    /* using shapes polymorphically */
    Shape * shapes[2];
    shapes[0] = new Rectangle (10, 20, 5, 6);
    shapes[1] = new Circle (15, 25, 8);

    for (int i=0; i<2; ++i)
    {
        shapes[i]->Draw();
        DoSomethingWithShape (shapes[i]);
    }

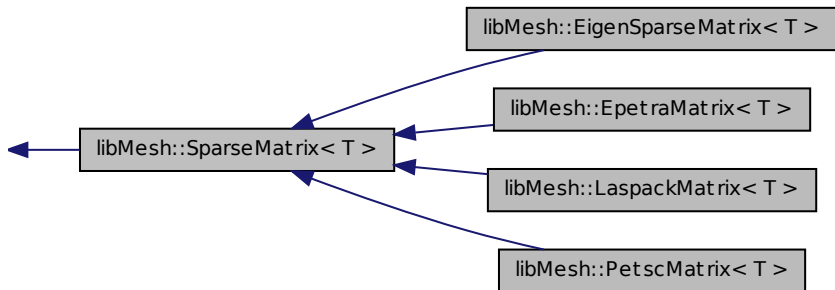
    /* access a rectangle specific function */
    Rectangle rect(0, 0, 15, 15);
    rect.SetWidth (30);
    rect.Draw ();
    ...
}
```



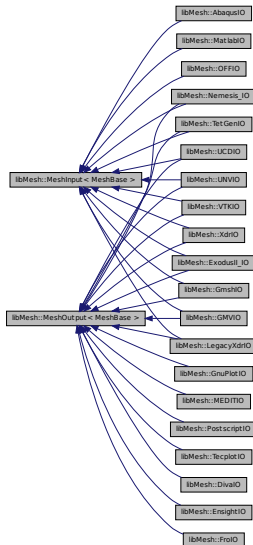
Examples of Polymorphism in **libMesh**



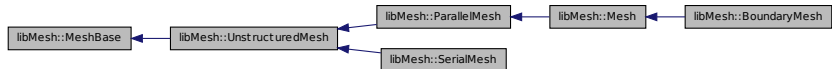
The “Glue:” Linear Algebra



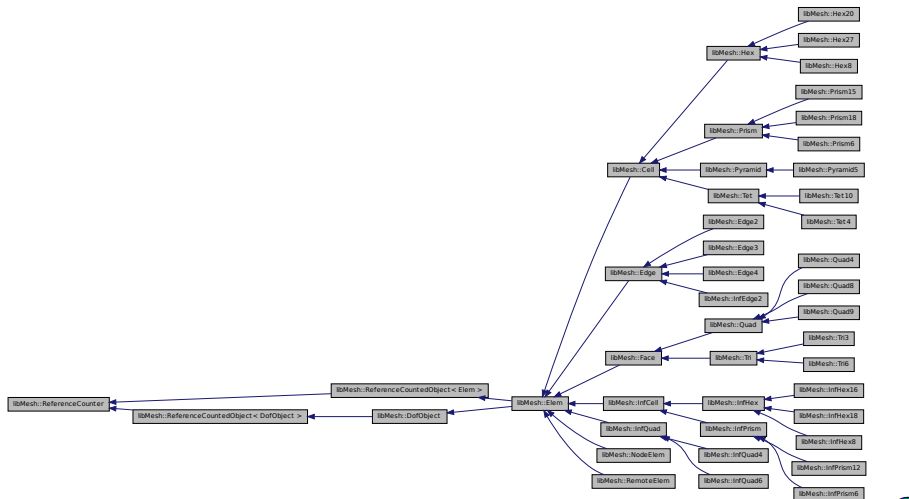
The “Glue:” I/O formats



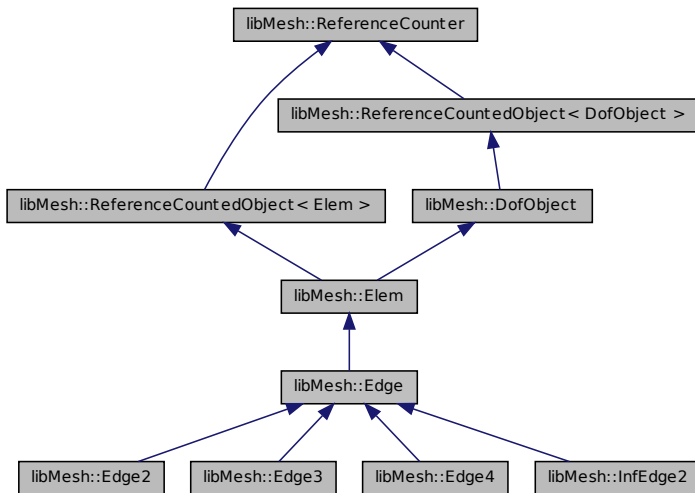
Discretization: The Mesh



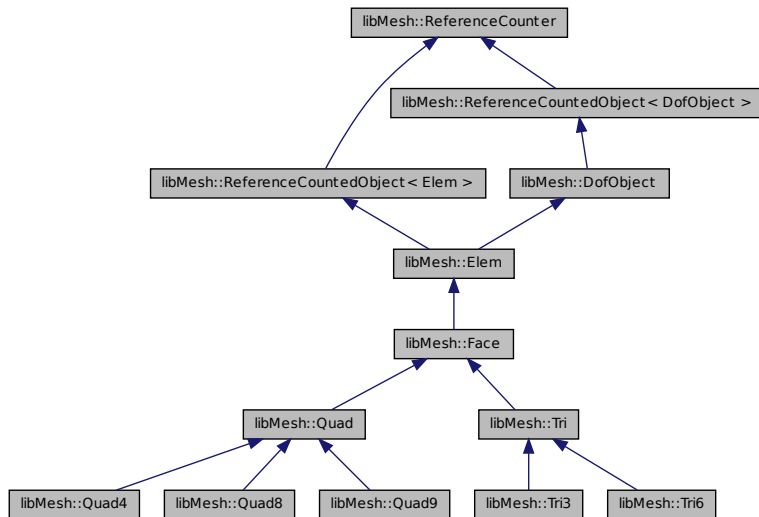
Disretization: Geometric Elements



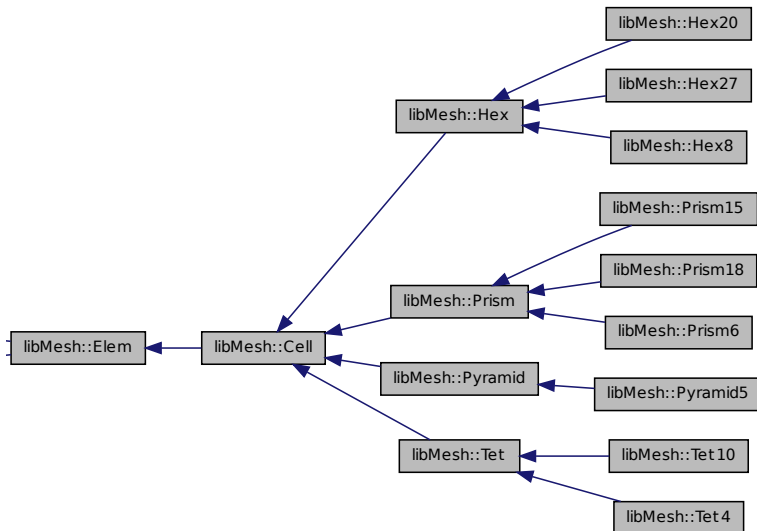
Discretization: Geometric Elements



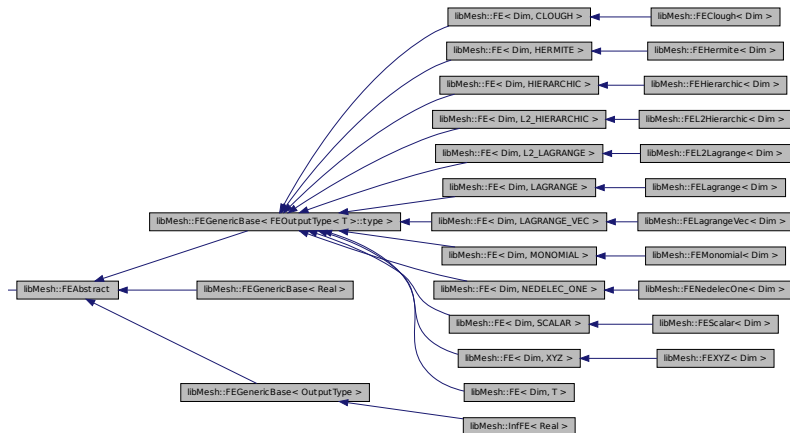
Discretization: Geometric Elements



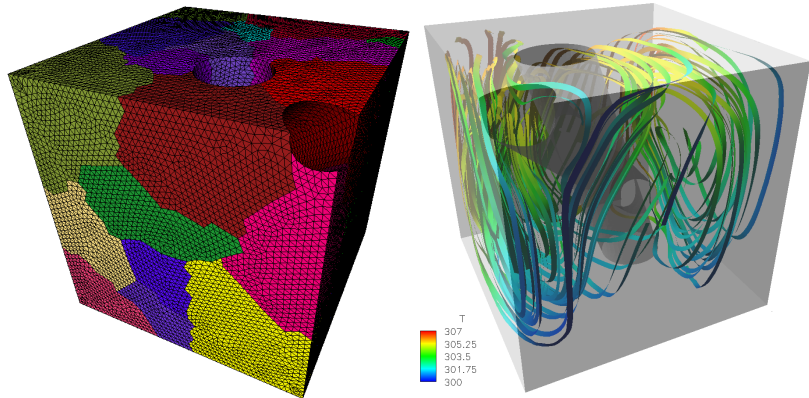
Disretization: Geometric Elements



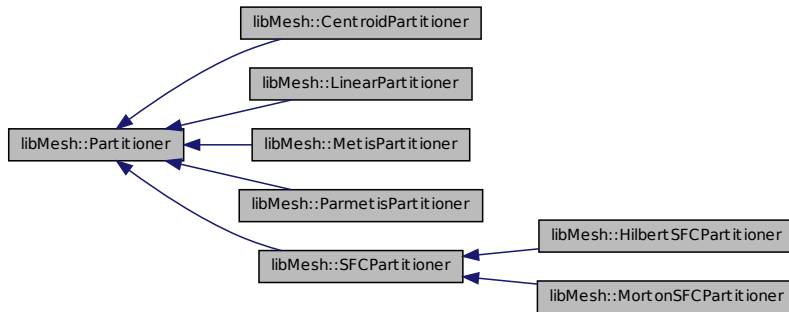
Disretization: Finite Elements



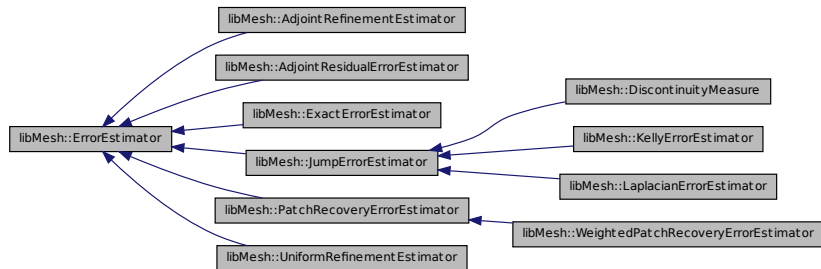
Algorithms: Domain Partitioning



Algorithms: Domain Partitioning



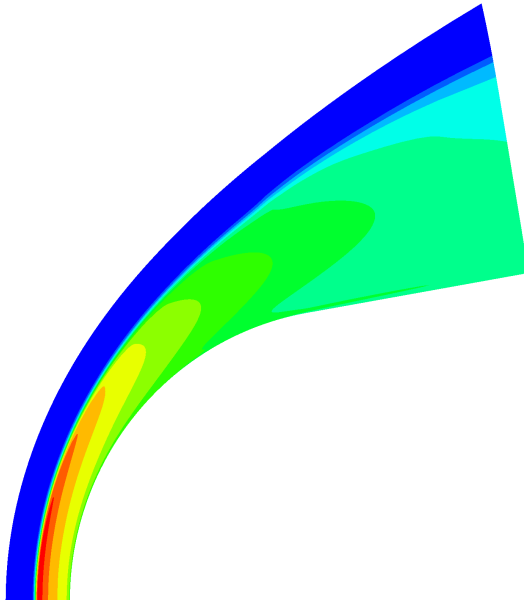
Algorithms: Error Estimation



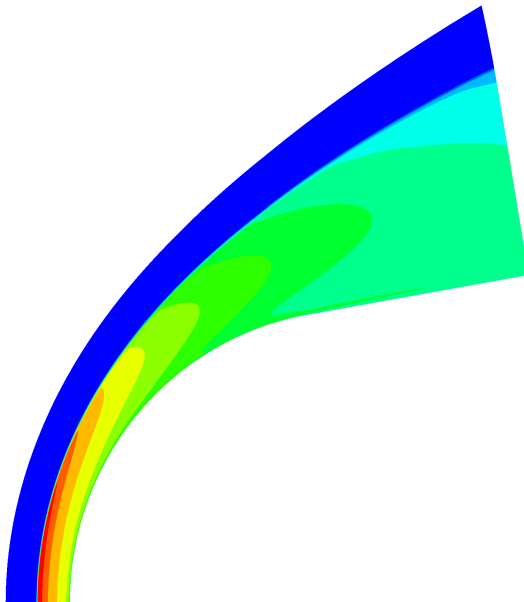
Results from Physics Applications built on top of **libMesh**



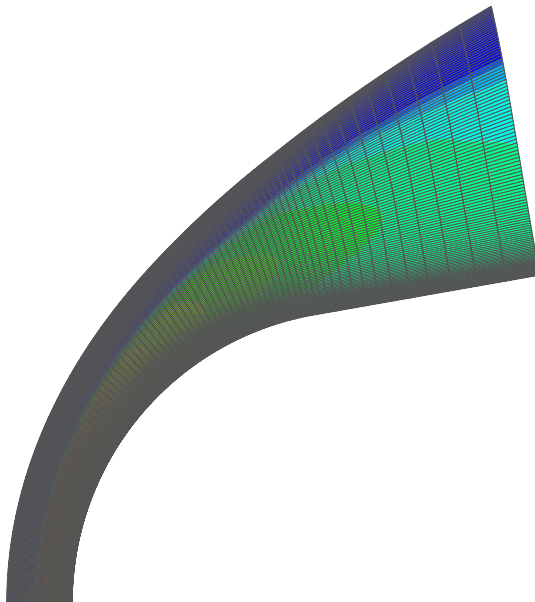
Compressible Navier-Stokes



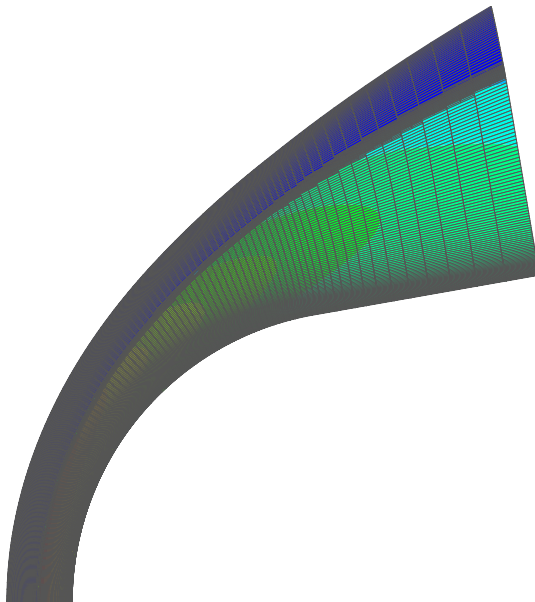
Compressible Navier-Stokes



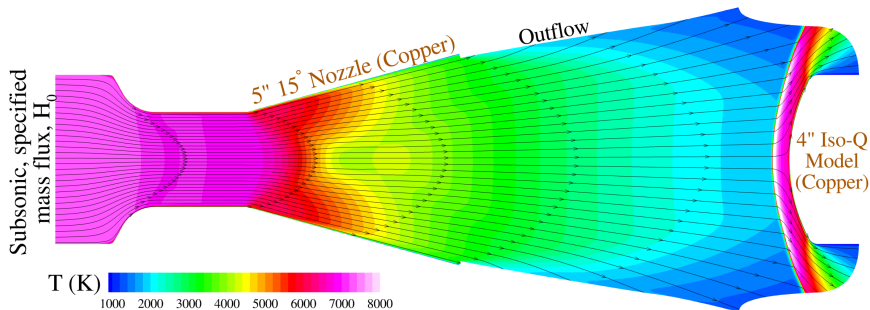
Compressible Navier-Stokes



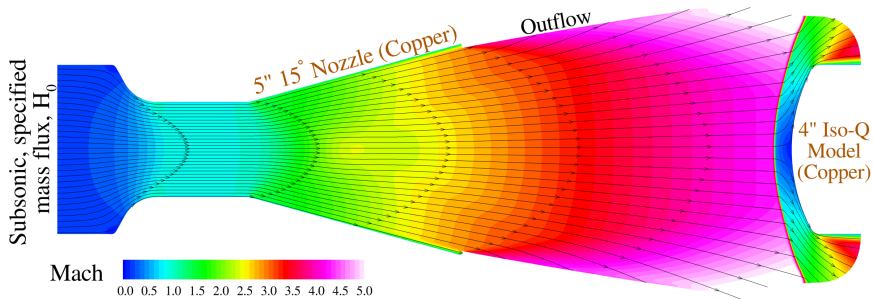
Compressible Navier-Stokes



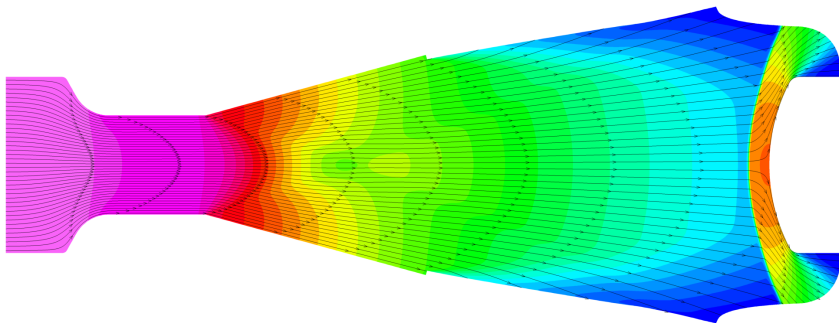
Arcjet Nozzle Calculation



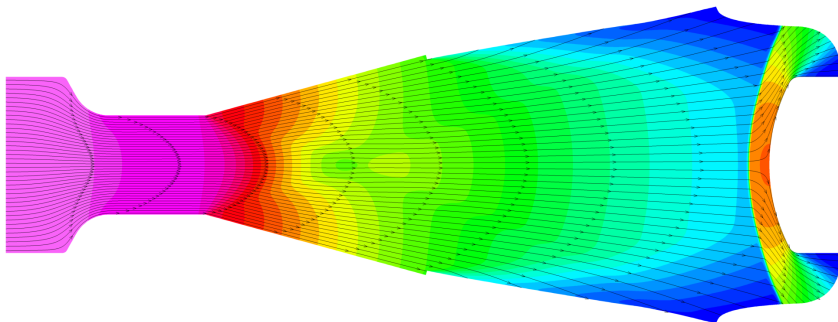
Arcjet Nozzle Calculation



Arcjet Nozzle Calculation

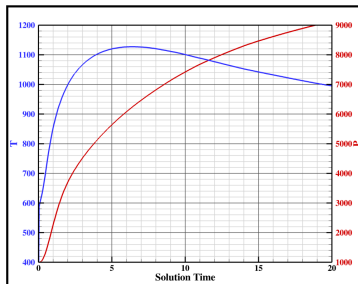
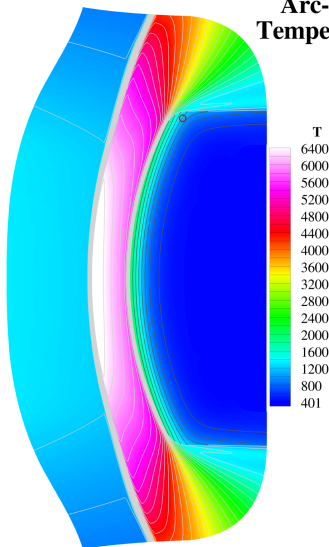


Arcjet Nozzle Calculation



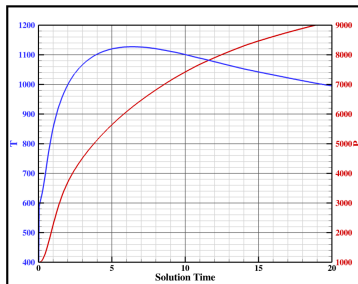
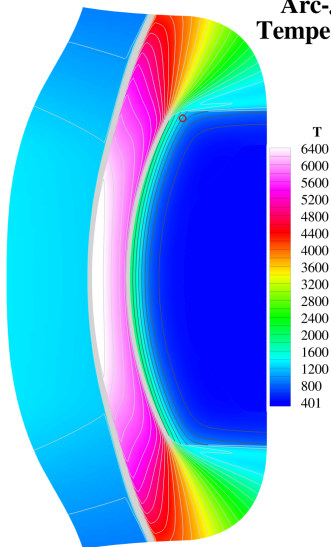
Coupled Pyrolysis, Temperature

Arc-Jet 4" Iso-Q Temperature History

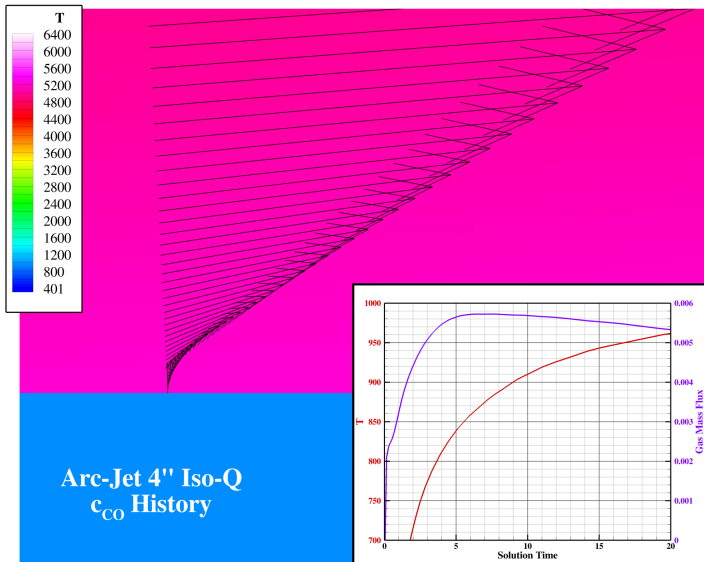


Coupled Pyrolysis, Temperature

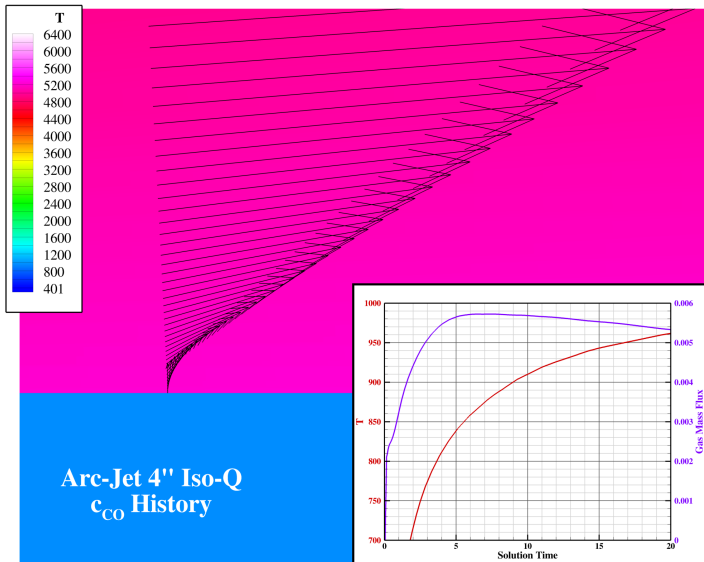
Arc-Jet 4" Iso-Q Temperature History



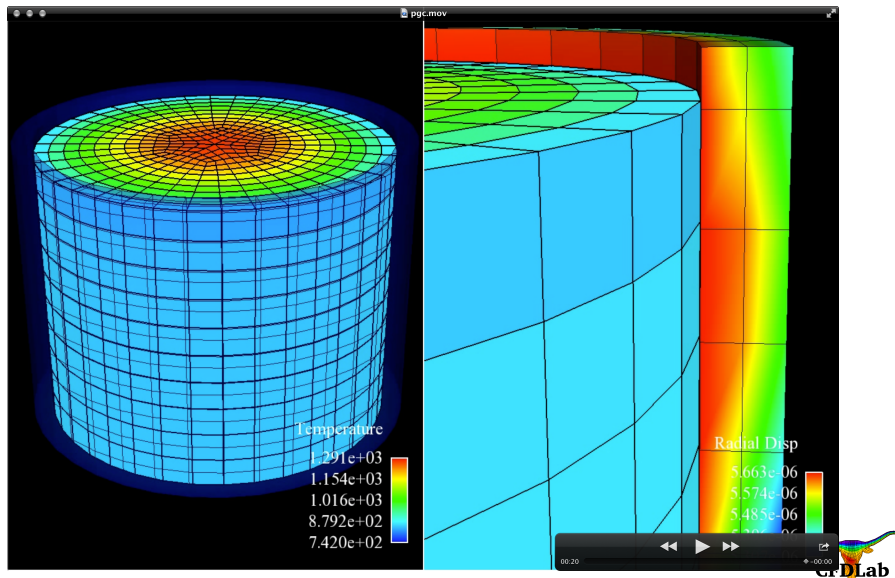
Coupled Pyrolysis, Pyrolysis gas mass flux, \dot{m}



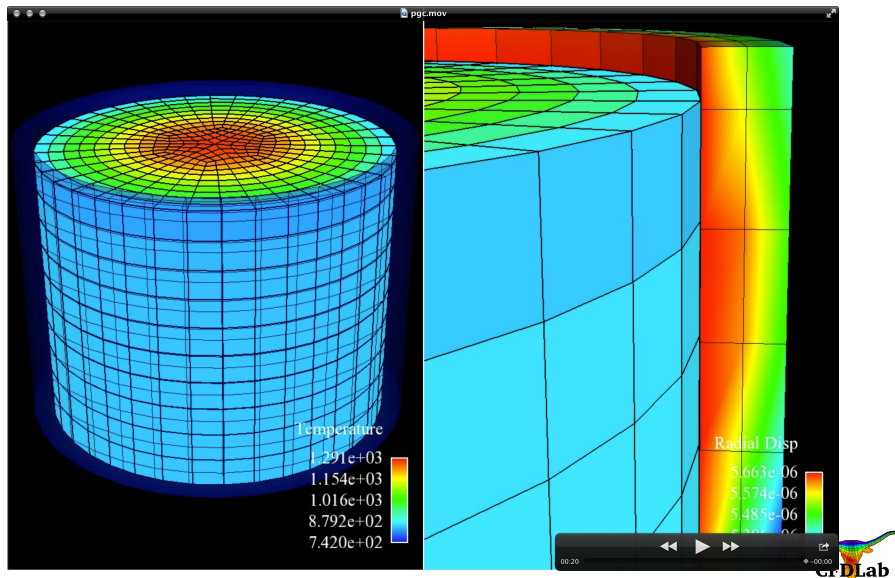
Coupled Pyrolysis, Pyrolysis gas mass flux, \dot{m}




Coupled Thermal/Solid Mechanics



Coupled Thermal/Solid Mechanics



The MOOSE Framework - Gaston et al., INL



www.inl.gov

INL
Idaho National
Laboratory

MOOSE

Derek Gaston, Idaho National Laboratory

Collaborators:

- Dana Knoll, LANL**
- Glen Hansen, INL**
- Chris Newman, LANL**
- Ryosuke Park, INL**
- Cody Permann, INL**
- David Andrs, INL**
- Hai Huang, INL**
- Michael Tonks, INL**
- Robert Podgorney, INL**

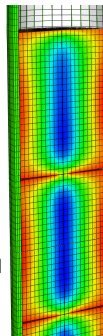


The MOOSE Framework - Gaston et al., INL




MOOSE – Multiphysics Object Oriented Simulation Environment

- A framework for solving computational nuclear engineering problems in a well planned, managed, and coordinated way
 - *Leveraged across multiple programs*
- Designed to significantly reduce the expense and time required to develop new applications
- Designed to develop analysis tools
 - *Uses very robust solution methods*
 - *Designed to be easily extended and maintained*
 - *Efficient on both a few and many processors*
- Currently supports ~7 applications which are developed and used by ~20 scientists.



The MOOSE Framework - Gaston et al., INL

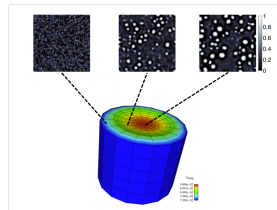
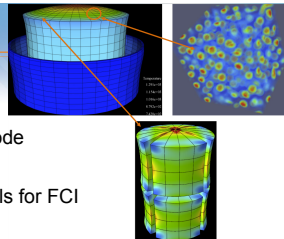
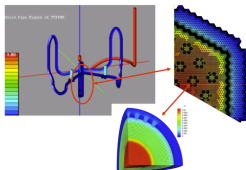
<div> MOOSE Ecosystem  </div>				
Application	Physics	Start	Time To Results	Lines of Code
BISON	Thermo-mechanics, Chemical Diffusion, coupled mesoscale	June 2008	4 Months	931
PRONGHORN	Neutronics, Porous Flow, Eigenvalue	September 2008	3 Months	2,883
SALMON	Multiphase Porous Flow	June 2009	3 Months	800
MARMOT	4 th Order Phasefield Mesoscale	August 2009	1 Month	838
RAT	Porous ReActive Transport	August 2009	1 Month	439
FALCON	Geo-mechanics, coupled mesoscale	September 2009	3 Months	810



The MOOSE Framework - Gaston et al., INL

BISON fuel performance

- LWR, Triso, and TRU fuel performance code
- Parallel 1D-3D thermomechanics code
- Thermal, mechanical, and chemical models for FCI
- Constituent redistribution
- Material, fission product swelling, fission gas release models
- Mesoscale-informed material models



Sidebar:

Revision Control & Collaboration with GitHub

Continuous Integration with Buildbot



Solving Problems the **libMesh** way

Discretizing a Generic Boundary Value Problem



- We assume there is a Boundary Value Problem of the form

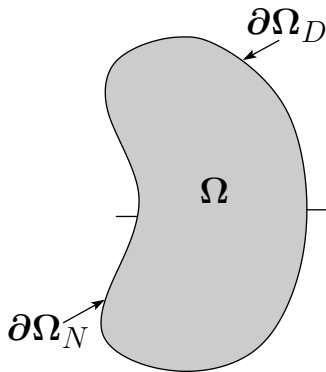
$$M \frac{\partial u}{\partial t} = F(u) \quad \in \Omega$$

$$G(u) = 0 \quad \in \Omega$$

$$u = u_D \quad \in \partial\Omega_D$$

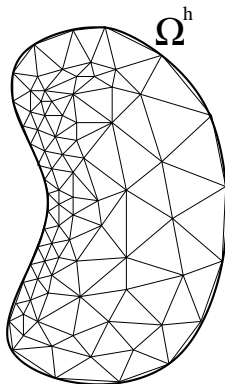
$$N(u) = 0 \quad \in \partial\Omega_N$$

$$u(\mathbf{x}, 0) = u_0(\mathbf{x})$$



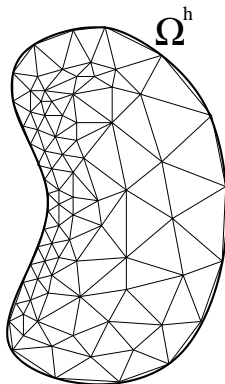
- Associated to the problem domain Ω is a `libMesh` data structure called a `Mesh`
- A `Mesh` is essentially a collection of finite elements

$$\Omega^h := \bigcup_e \Omega_e$$



- Associated to the problem domain Ω is a `libMesh` data structure called a `Mesh`
- A `Mesh` is essentially a collection of finite elements

$$\Omega^h := \bigcup_e \Omega_e$$



- `libMesh` provides some simple structured mesh generation routines, file inputs, and interfaces to Triangle and TetGen.

The Mesh

```
int main (int argc, char** argv)
{
    // Initialize the library.  This is necessary because the library
    // may depend on a number of other libraries (i.e. MPI and PETSc)
    // that require initialization before use.  When the LibMeshInit
    // object goes out of scope, other libraries and resources are
    // finalized.
    LibMeshInit init (argc, argv);

    // Create a mesh
    Mesh mesh;

    // Read the input mesh.
    mesh.read ("in.exo");

    // Print information about the mesh to the screen.
    mesh.print_info();

    // Write the output mesh
    mesh.write ("out.dat");

    ...
}
```



The Mesh

```
*****
* Running Example introduction_ex1:
*   example-opt -d 3 reference_elements/3D/one_hex27.xda -o output.xda
*****
```

Mesh Information:

```
mesh_dimension()=3
spatial_dimension()=3
n_nodes()=27
  n_local_nodes()=27
n_elem()=1
  n_local_elem()=1
  n_active_elem()=1
n_subdomains()=1
n_partitions()=1
n_processors()=1
n_threads()=1
processor_id()=0
```



Mesh Iterators

```
void foo (const MeshBase &mesh)
{
    // Now we will loop over all the elements in the mesh that
    // live on the local processor. We will compute the element
    // matrix and right-hand-side contribution. Since the mesh
    // may be refined we want to only consider the ACTIVE elements,
    // hence we use a variant of the \p active_elem_iterator.
    MeshBase::const_element_iterator
        el      = mesh.active_local_elements_begin();
    const MeshBase::const_element_iterator
        end_el  = mesh.active_local_elements_end();

    for ( ; el != end_el; ++el)
    {
        // Store a pointer to the element we are currently
        // working on. This allows for nicer syntax later.
        const Elem* elem = *el;
        ...
    }
    ...
}
```



Mesh Iterators

```
void foo (const MeshBase &mesh)
{
    // We will now loop over all nodes.
    MeshBase::const_node_iterator    node_it  = mesh.nodes_begin();
    const MeshBase::const_node_iterator node_end = mesh.nodes_end();

    for ( ; node_it != node_end; ++node_it)
    {
        // the current node pointer
        const Node* node = *node_it;
        ...
    }
    ...
}
```



```
...
// Create a mesh.
Mesh mesh;

// Create a uniform 5x5 mesh on the unit square.
MeshTools::Generation::build_square (mesh, 5, 5);
mesh.print_info();

// Create an equation systems object. This object can contain
// multiple systems of different flavors for solving loosely coupled
// physics. Each system can contain multiple variables of different
// approximation orders. The EquationSystems object needs a
// reference to the mesh object, so the order of construction here
// is important.
EquationSystems equation_systems (mesh);

// Now we declare the system and its variables. We begin by adding
// a "TransientLinearImplicitSystem" to the EquationSystems object,
// and we give it the name "Simple System".
equation_systems.add_system<TransientLinearImplicitSystem> ("Simple System");

// Adds the variable "u" to "Simple System". "u" will be
// approximated using first-order approximation.
equation_systems.get_system("Simple System").add_variable("u", FIRST);

// Next we'll add an "ExplicitSystem" to the EquationSystems
// object, and we give it the name "Complex System".
equation_systems.add_system<ExplicitSystem> ("Complex System");

// Give "Complex System" three variables -- each with a different
// approximation order. Variables "c" and "T" will use first-order
// Lagrange approximation, while variable "dv" will use a
// second-order discontinuous approximation space.
equation_systems.get_system("Complex System").add_variable("c", FIRST);
equation_systems.get_system("Complex System").add_variable("T", FIRST);
equation_systems.get_system("Complex System").add_variable("dv", SECOND, MONOMIAL);

// Initialize the data structures for the equation system.
equation_systems.init();

// Prints information about the system to the screen.
equation_systems.print_info();
...
```



- 1 Introduction
 - Background
 - The `libMesh` Software Library
 - Software Reusability
 - Library Design
- 2 Motivation
 - Application Results
- 3 Approach to Software Development
- 4 A Generic Boundary Value Problem
- 5 Key Data Structures
 - The Mesh Class
 - The EquationSystems Class
- 6 Weighted Residuals**
- 7 Poisson Equation
- 8 Other Examples
- 9 Some Extensions



- The point of departure in any FE analysis which uses `libMesh` is the weighted residual statement

$$(F(u), v) = 0 \quad \forall v \in \mathcal{V}$$



- The point of departure in any FE analysis which uses `libMesh` is the weighted residual statement

$$(F(u), v) = 0 \quad \forall v \in \mathcal{V}$$

- Or, more precisely, the weighted residual statement associated with the finite-dimensional space $\mathcal{V}^h \subset \mathcal{V}$

$$(F(u^h), v^h) = 0 \quad \forall v^h \in \mathcal{V}^h$$



Poisson Equation

$$-\Delta u = f \quad \in \quad \Omega$$



Poisson Equation

$$-\Delta u = f \quad \in \quad \Omega$$

Weighted Residual Statement

$$\begin{aligned} (F(u), v) &:= \int_{\Omega} [\nabla u \cdot \nabla v - fv] \, dx \\ &+ \int_{\partial\Omega_N} (\nabla u \cdot \mathbf{n}) v \, ds \end{aligned}$$



Linear Convection-Diffusion

$$-k\Delta u + \mathbf{b} \cdot \nabla u = f \quad \in \quad \Omega$$



Linear Convection-Diffusion

$$-k\Delta u + \mathbf{b} \cdot \nabla u = f \quad \in \quad \Omega$$

Weighted Residual Statement

$$\begin{aligned} (F(u), v) := & \int_{\Omega} [k \nabla u \cdot \nabla v + (\mathbf{b} \cdot \nabla u) v - f v] dx \\ & + \int_{\partial\Omega_N} k (\nabla u \cdot \mathbf{n}) v ds \end{aligned}$$



Stokes Flow

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega$$



Stokes Flow

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega$$

Weighted Residual Statement

$$\mathbf{u} := [\mathbf{u}, p] \quad , \quad \mathbf{v} := [\mathbf{v}, q]$$

$$\begin{aligned}(F(\mathbf{u}), \mathbf{v}) &:= \int_{\Omega} [-p (\nabla \cdot \mathbf{v}) + \nu \nabla \mathbf{u} : \nabla \mathbf{v} - \mathbf{f} \cdot \mathbf{v} \\ &\quad + (\nabla \cdot \mathbf{u}) q] dx + \int_{\partial \Omega_N} (\nu \nabla \mathbf{u} - p \mathbf{I}) \mathbf{n} \cdot \mathbf{v} ds\end{aligned}$$



- To obtain the approximate problem, we simply replace $u \leftarrow u^h$, $v \leftarrow v^h$, and $\Omega \leftarrow \Omega^h$ in the weighted residual statement.



- 1 Introduction
 - Background
 - The `libMesh` Software Library
 - Software Reusability
 - Library Design
- 2 Motivation
 - Application Results
- 3 Approach to Software Development
- 4 A Generic Boundary Value Problem
- 5 Key Data Structures
 - The Mesh Class
 - The `EquationSystems` Class
- 6 Weighted Residuals
- 7 Poisson Equation**
- 8 Other Examples
- 9 Some Extensions



- For simplicity we start with the weighted residual statement arising from the Poisson equation, with $\partial\Omega_N = \emptyset$,

$$(F(u^h), v^h) := \int_{\Omega^h} [\nabla u^h \cdot \nabla v^h - f v^h] dx = 0 \quad \forall v^h \in \mathcal{V}^h$$



- The integral over Ω^h ...

$$0 = \int_{\Omega^h} [\nabla u^h \cdot \nabla v^h - f v^h] dx \quad \forall v^h \in \mathcal{V}^h$$



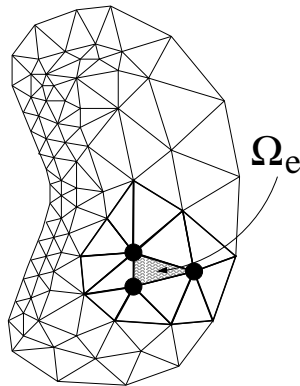
- The integral over Ω^h ... is written as a sum of integrals over the N_e finite elements:

$$\begin{aligned} 0 &= \int_{\Omega^h} [\nabla u^h \cdot \nabla v^h - f v^h] dx \quad \forall v^h \in \mathcal{V}^h \\ &= \sum_{e=1}^{N_e} \int_{\Omega_e} [\nabla u^h \cdot \nabla v^h - f v^h] dx \quad \forall v^h \in \mathcal{V}^h \end{aligned}$$



- An element integral will have contributions only from the global basis functions corresponding to its nodes.
- We call these local basis functions ϕ_i , $0 \leq i \leq N_s$.

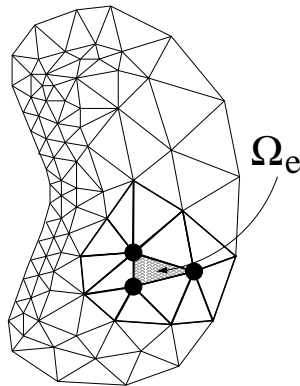
$$v^h|_{\Omega_e} = \sum_{i=1}^{N_s} c_i \phi_i$$



- An element integral will have contributions only from the global basis functions corresponding to its nodes.
- We call these local basis functions ϕ_i , $0 \leq i \leq N_s$.

$$v^h|_{\Omega_e} = \sum_{i=1}^{N_s} c_i \phi_i$$

$$\int_{\Omega_e} v^h dx = \sum_{i=1}^{N_s} c_i \int_{\Omega_e} \phi_i dx$$



- The element integrals ...

$$\int_{\Omega_e} [\nabla u^h \cdot \nabla v^h - f v^h] dx$$



- The element integrals ...

$$\int_{\Omega_e} [\nabla u^h \cdot \nabla v^h - f v^h] dx$$

- are written in terms of the local “ ϕ_i ” basis functions

$$\sum_{j=1}^{N_s} u_j \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i dx - \int_{\Omega_e} f \phi_i dx \quad , \quad i = 1, \dots, N_s$$



- The element integrals ...

$$\int_{\Omega_e} [\nabla u^h \cdot \nabla v^h - f v^h] dx$$

- are written in terms of the local “ ϕ_i ” basis functions

$$\sum_{j=1}^{N_s} u_j \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i dx - \int_{\Omega_e} f \phi_i dx \quad , \quad i = 1, \dots, N_s$$

- This can be expressed naturally in matrix notation as

$$\mathbf{K}^e \mathbf{U}^e - \mathbf{F}^e$$



- The entries of the element stiffness matrix are the integrals

$$\mathbf{K}_{ij}^e := \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i \, dx$$



- The entries of the element stiffness matrix are the integrals

$$\mathbf{K}_{ij}^e := \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i \, dx$$

- While for the element right-hand side we have

$$\mathbf{F}_i^e := \int_{\Omega_e} f \phi_i \, dx$$



- The entries of the element stiffness matrix are the integrals

$$\mathbf{K}_{ij}^e := \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i \, dx$$

- While for the element right-hand side we have

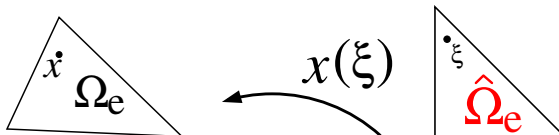
$$\mathbf{F}_i^e := \int_{\Omega_e} f \phi_i \, dx$$

- The element stiffness matrices and right-hand sides can be “assembled” to obtain the global system of equations

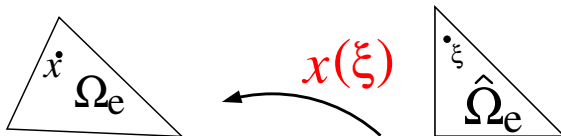
$$\mathbf{K}\mathbf{U} = \mathbf{F}$$



- The integrals are performed on a “reference” element $\hat{\Omega}_e$



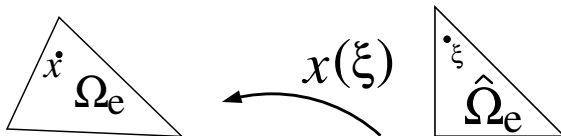
- The integrals are performed on a “reference” element $\hat{\Omega}_e$



- The Jacobian of the map $x(\xi)$ is J .

$$\mathbf{F}_i^e = \int_{\Omega_e} f \phi_i dx = \int_{\hat{\Omega}_e} f(x(\xi)) \phi_i |J| d\xi$$

- The integrals are performed on a “reference” element $\hat{\Omega}_e$



- Chain rule: $\nabla = J^{-1} \nabla_{\xi} := \hat{\nabla}_{\xi}$

$$K_{ij}^e = \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i \, dx = \int_{\hat{\Omega}_e} \hat{\nabla}_{\xi} \phi_j \cdot \hat{\nabla}_{\xi} \phi_i \, |J| \, d\xi$$

- The integrals on the “reference” element are approximated via numerical quadrature.



- The integrals on the “reference” element are approximated via numerical quadrature.
- The quadrature rule has N_q points “ ξ_q ” and weights “ w_q ”.



- The integrals on the “reference” element are approximated via numerical quadrature.
- The quadrature rule has N_q points “ ξ_q ” and weights “ w_q ”.

$$\begin{aligned} \mathbf{F}_i^e &= \int_{\hat{\Omega}_e} f \phi_i |J| d\xi \\ &\approx \sum_{q=1}^{N_q} f(x(\xi_q)) \phi_i(\xi_q) |J(\xi_q)| w_q \end{aligned}$$



- The integrals on the “reference” element are approximated via numerical quadrature.
- The quadrature rule has N_q points “ ξ_q ” and weights “ w_q ”.

$$\begin{aligned} \mathbf{K}_{ij}^e &= \int_{\hat{\Omega}_e} \hat{\nabla}_{\xi} \phi_j \cdot \hat{\nabla}_{\xi} \phi_i |J| d\xi \\ &\approx \sum_{q=1}^{N_q} \hat{\nabla}_{\xi} \phi_j(\xi_q) \cdot \hat{\nabla}_{\xi} \phi_i(\xi_q) |J(\xi_q)| w_q \end{aligned}$$



- libMesh provides the following variables at each quadrature point q

Code	Math	Description
JxW[q]	$ J(\xi_q) w_q$	Jacobian times weight
phi[i][q]	$\phi_i(\xi_q)$	value of i^{th} shape fn.
dphi[i][q]	$\hat{\nabla}_\xi \phi_i(\xi_q)$	value of i^{th} shape fn. gradient
xyz[q]	$x(\xi_q)$	location of ξ_q in physical space



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i) += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$F_i^e = \sum_{q=1}^{N_q} f(x(\xi_q)) \phi_i(\xi_q) |J(\xi_q)| w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$F_i^e = \sum_{q=1}^{N_q} f(x(\xi_q)) \phi_i(\xi_q) |J(\xi_q)| w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i) += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$F_i^e = \sum_{q=1}^{N_q} f(x(\xi_q)) \phi_i(\xi_q) |J(\xi_q)| w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i) += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$F_i^e = \sum_{q=1}^{N_q} f(x(\xi_q)) \phi_i(\xi_q) |J(\xi_q)| w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$K_{ij}^e = \sum_{q=1}^{N_q} \hat{\nabla}_{\xi} \phi_j(\xi_q) \cdot \hat{\nabla}_{\xi} \phi_i(\xi_q) |J(\xi_q)| w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```

for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }

```

$$K_{ij}^e = \sum_{q=1}^{N_q} \hat{\nabla}_{\xi} \phi_j(\xi_q) \cdot \hat{\nabla}_{\xi} \phi_i(\xi_q) |J(\xi_q)| w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i) += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$K_{ij}^e = \sum_{q=1}^{N_q} \hat{\nabla}_{\xi} \phi_j(\xi_q) \cdot \hat{\nabla}_{\xi} \phi_i(\xi_q) |J(\xi_q)| w_q$$



```
// We now define the matrix assembly function for the Poisson system
// by computing the element matrices and right-hand sides. We are
// omitting BCs for brevity.
void assemble_poisson(EquationSystems& es,
                     const std::string& system_name)
{
    // Get a constant reference to the mesh object.
    const MeshBase& mesh = es.get_mesh();

    // The dimension that we are running
    const unsigned int dim = mesh.mesh_dimension();

    // Get a reference to the LinearImplicitSystem we are solving
    LinearImplicitSystem& system = es.get_system<LinearImplicitSystem>("Poisson");

    // A reference to the DofMap object for this system. The DofMap
    // object handles the index translation from node and element
    // numbers to degree of freedom numbers. We will talk more about
    // the DofMap in future examples.
    const DofMap& dof_map = system.get_dof_map();

    // Get a constant reference to the Finite Element type for the first
    // (and only) variable in the system.
    FEType fe_type = dof_map.variable_type(0);

    // Build a Finite Element object of the specified type. Since the
    // FEBase::build() member dynamically creates memory we will store
    // the object as an AutoPtr<FEBase>. This can be thought of as a
    // pointer that will clean up after itself.
    AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));

    // A 5th order Gauss quadrature rule for numerical integration.
    QGauss qrule (dim, FIFTH);
```



```

// Tell the finite element object to use our quadrature rule.
fe->attach_quadrature_rule (&qrule);

// Here we define some references to cell-specific data that will be
// used to assemble the linear system. We begin with the element
// Jacobian * quadrature weight at each integration point.
const std::vector<Real>& JxW = fe->get_JxW();

// The physical XY locations of the quadrature points on the
// element. These might be useful for evaluating spatially varying
// material properties at the quadrature points.
const std::vector<Point>& q_point = fe->get_xyz();

// The element shape functions evaluated at the quadrature points.
const std::vector<std::vector<Real> >& phi = fe->get_phi();

// The element shape function gradients evaluated at the quadrature
// points.
const std::vector<std::vector<RealGradient> >& dphi = fe->get_dphi();

// Define data structures to contain the element matrix and
// right-hand-side vector contribution. Following basic finite
// element terminology we will denote these "Ke" and "Fe".
DenseMatrix<Number> Ke;
DenseVector<Number> Fe;

// This vector will hold the degree of freedom indices for the
// element. These define where in the global system the element
// degrees of freedom get mapped.
std::vector<dof_id_type> dof_indices;

// Now we will loop over all the elements in the mesh. We will

```




```
// compute the element matrix and right-hand-side contribution.  See
// example 3 for a discussion of the element iterators.
MeshBase::const_element_iterator    el      = mesh.active_local_elements_begin();
const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();

for ( ; el != end_el; ++el)
{
    // Store a pointer to the element we are currently working on.
    // This allows for nicer syntax later.
    const Elem* elem = *el;

    // Get the degree of freedom indices for the current element.
    // These define where in the global matrix and right-hand-side
    // this element will contribute to.
    dof_map.dof_indices (elem, dof_indices);

    // Compute the element-specific data for the current element.
    // This involves computing the location of the quadrature points
    // (q_point) and the shape functions (phi, dphi) for the current
    // element.
    fe->reinit (elem);

    // Zero the element matrix and right-hand side before summing
    // them.  We use the resize member here because the number of
    // degrees of freedom might have changed from the last element.
    // Note that this will be the case if the element type is
    // different (i.e. the last element was a triangle, now we are
    // on a quadrilateral).
    Ke.resize (dof_indices.size(),
              dof_indices.size());

    Fe.resize (dof_indices.size());
```



```

// Now we will build the element matrix.  This involves a double
// loop to integrate the test functions (i) against the trial
// functions (j).
for (unsigned int qp=0; qp<qrule.n_points(); qp++)
    for (unsigned int i=0; i<phi.size(); i++)
        for (unsigned int j=0; j<phi.size(); j++)
            Ke(i,j) += JxW[qp]*(dphi[i][qp]*dphi[j][qp]);

// Now we build the element right-hand-side contribution.  This
// involves a single loop in which we integrate the "forcing
// function" in the PDE against the test functions.
for (unsigned int qp=0; qp<qrule.n_points(); qp++)
    for (unsigned int i=0; i<phi.size(); i++)
        Fe(i) += JxW[qp]*10.*phi[i][qp];

// If we are using an adaptive mesh this will apply any hanging
// node constraint equations
dof_map.heterogenously_constrain_element_matrix_and_vector (Ke, Fe, dof_indices);

// The element matrix and right-hand-side are now built for this
// element.  Add them to the global matrix and right-hand-side
// vector.  The SparseMatrix::add_matrix() and
// NumericVector::add_vector() members do this for us.
system.matrix->add_matrix (Ke, dof_indices);
system.rhs->add_vector (Fe, dof_indices);
}
}

```



- 1 Introduction
 - Background
 - The `libMesh` Software Library
 - Software Reusability
 - Library Design
- 2 Motivation
 - Application Results
- 3 Approach to Software Development
- 4 A Generic Boundary Value Problem
- 5 Key Data Structures
 - The Mesh Class
 - The EquationSystems Class
- 6 Weighted Residuals
- 7 Poisson Equation
- 8 Other Examples
- 9 Some Extensions



- The matrix assembly routine for the linear convection-diffusion equation,

$$-k\Delta u + \mathbf{b} \cdot \nabla u = f$$

```

for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(k*(dphi[j][q]*dphi[i][q])
                        + (b*dphi[j][q])*phi[i][q]);
  }

```



- The matrix assembly routine for the linear convection-diffusion equation,

$$-k\Delta u + \mathbf{b} \cdot \nabla u = f$$

```

for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(k*(dphi[j][q]*dphi[i][q])
                        + (b*dphi[j][q])*phi[i][q]);
  }

```



- The matrix assembly routine for the linear convection-diffusion equation,

$$-k\Delta u + \mathbf{b} \cdot \nabla u = f$$

```

for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(k*(dphi[j][q]*dphi[i][q])
                        + (b*dphi[j][q])*phi[i][q]);
  }

```



- For multi-variable systems like Stokes flow,

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can be extended to include sub-matrices

$$\left[\begin{array}{cc|c} K_{u_1 u_1}^e & K_{u_1 u_2}^e & K_{u_1 p}^e \\ K_{u_2 u_1}^e & K_{u_2 u_2}^e & K_{u_2 p}^e \\ \hline K_{p u_1}^e & K_{p u_2}^e & K_{p p}^e \end{array} \right] \left[\begin{array}{c} U_{u_1}^e \\ U_{u_2}^e \\ U_p^e \end{array} \right] = \left[\begin{array}{c} F_{u_1}^e \\ F_{u_2}^e \\ F_p^e \end{array} \right]$$

- We have an array of submatrices: $\mathbf{K}^e = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$



- For multi-variable systems like Stokes flow,

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can be extended to include sub-matrices

$$\left[\begin{array}{cc|c} K_{u_1 u_1}^e & K_{u_1 u_2}^e & K_{u_1 p}^e \\ K_{u_2 u_1}^e & K_{u_2 u_2}^e & K_{u_2 p}^e \\ \hline K_{p u_1}^e & K_{p u_2}^e & K_{p p}^e \end{array} \right] \left[\begin{array}{c} U_{u_1}^e \\ U_{u_2}^e \\ U_p^e \end{array} \right] = \left[\begin{array}{c} F_{u_1}^e \\ F_{u_2}^e \\ F_p^e \end{array} \right]$$

- We have an array of submatrices: $\mathbf{K}_e [0] [0]$



- For multi-variable systems like Stokes flow,

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can be extended to include sub-matrices

$$\left[\begin{array}{cc|c} K_{u_1 u_1}^e & K_{u_1 u_2}^e & K_{u_1 p}^e \\ K_{u_2 u_1}^e & \textcolor{red}{K}_{u_2 u_2}^e & K_{u_2 p}^e \\ \hline K_{p u_1}^e & K_{p u_2}^e & K_{p p}^e \end{array} \right] \left[\begin{array}{c} U_{u_1}^e \\ U_{u_2}^e \\ U_p^e \end{array} \right] = \left[\begin{array}{c} F_{u_1}^e \\ F_{u_2}^e \\ F_p^e \end{array} \right]$$

- We have an array of submatrices: $\mathbf{K}_e [1] [1]$



- For multi-variable systems like Stokes flow,

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can be extended to include sub-matrices

$$\left[\begin{array}{cc|c} K_{u_1 u_1}^e & K_{u_1 u_2}^e & K_{u_1 p}^e \\ K_{u_2 u_1}^e & K_{u_2 u_2}^e & K_{u_2 p}^e \\ \hline K_{p u_1}^e & \mathbf{K}_{p u_2}^e & K_{p p}^e \end{array} \right] \left[\begin{array}{c} U_{u_1}^e \\ U_{u_2}^e \\ U_p^e \end{array} \right] = \left[\begin{array}{c} F_{u_1}^e \\ F_{u_2}^e \\ F_p^e \end{array} \right]$$

- We have an array of submatrices: $\mathbf{K}_e [2] [1]$



- For multi-variable systems like Stokes flow,

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can be extended to include sub-matrices

$$\left[\begin{array}{cc|c} K_{u_1 u_1}^e & K_{u_1 u_2}^e & K_{u_1 p}^e \\ K_{u_2 u_1}^e & K_{u_2 u_2}^e & K_{u_2 p}^e \\ \hline K_{p u_1}^e & K_{p u_2}^e & K_{p p}^e \end{array} \right] \left[\begin{array}{c} U_{u_1}^e \\ U_{u_2}^e \\ U_p^e \end{array} \right] = \left[\begin{array}{c} F_{u_1}^e \\ F_{u_2}^e \\ F_p^e \end{array} \right]$$

- And an array of right-hand sides: $\mathbf{F}^e []$.

- For multi-variable systems like Stokes flow,

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can be extended to include sub-matrices

$$\left[\begin{array}{cc|c} K_{u_1 u_1}^e & K_{u_1 u_2}^e & K_{u_1 p}^e \\ K_{u_2 u_1}^e & K_{u_2 u_2}^e & K_{u_2 p}^e \\ \hline K_{p u_1}^e & K_{p u_2}^e & K_{p p}^e \end{array} \right] \left[\begin{array}{c} U_{u_1}^e \\ U_{u_2}^e \\ U_p^e \end{array} \right] = \left[\begin{array}{c} F_{u_1}^e \\ F_{u_2}^e \\ F_p^e \end{array} \right]$$

- And an array of right-hand sides: $\mathbf{Fe}[0]$.



- For multi-variable systems like Stokes flow,

$$\begin{aligned}\nabla p - \nu \Delta \mathbf{u} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0\end{aligned} \quad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can be extended to include sub-matrices

$$\left[\begin{array}{cc|c} K_{u_1 u_1}^e & K_{u_1 u_2}^e & K_{u_1 p}^e \\ K_{u_2 u_1}^e & K_{u_2 u_2}^e & K_{u_2 p}^e \\ \hline K_{p u_1}^e & K_{p u_2}^e & K_{p p}^e \end{array} \right] \left[\begin{array}{c} U_{u_1}^e \\ U_{u_2}^e \\ U_p^e \end{array} \right] = \left[\begin{array}{c} F_{u_1}^e \\ F_{u_2}^e \\ F_p^e \end{array} \right]$$

- And an array of right-hand sides: $\mathbf{F} \in [1]$.



- The matrix assembly can proceed in essentially the same way.
- For the momentum equations:

```
for (q=0; q<Nq; ++q)
  for (d=0; d<2; ++d)
    for (i=0; i<Ns; ++i) {
      Fe[d](i) += JxW[q]*f(xyz[q],d)*phi[i][q];

      for (j=0; j<Ns; ++j)
        Ke[d][d](i,j) +=
          JxW[q]*nu*(dphi[j][q]*dphi[i][q]);
    }
```



- 1 Introduction
 - Background
 - The `libMesh` Software Library
 - Software Reusability
 - Library Design
- 2 Motivation
 - Application Results
- 3 Approach to Software Development
- 4 A Generic Boundary Value Problem
- 5 Key Data Structures
 - The Mesh Class
 - The EquationSystems Class
- 6 Weighted Residuals
- 7 Poisson Equation
- 8 Other Examples
- 9 Some Extensions



- For linear problems, we have already seen how the weighted residual statement leads directly to a sparse linear system of equations

$$KU = F$$



- For time-dependent problems,

$$\frac{\partial u}{\partial t} = F(u)$$

- we also need a way to advance the solution in time, e.g. a θ -method

$$\left(\frac{u^{n+1} - u^n}{\Delta t}, v^h \right) = (F(u_\theta), v^h) \quad \forall v^h \in \mathcal{V}^h$$
$$u_\theta := \theta u^{n+1} + (1 - \theta) u^n$$

- Leads to $KU = F$ at *each timestep*.



- For nonlinear problems, typically a sequence of linear problems must be solved, e.g. for Newton's method

$$(F'(u^k)\delta u^{k+1}, v) = -(F(u^k), v)$$

where $F'(u^k)$ is the linearized (Jacobian) operator associated with the PDE.

- Must solve $KU = F$ (Inexact Newton method) at *each iteration step*.



- B. Kirk, J. Peterson, R. Stogner and G. Carey, “libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations”, *Engineering with Computers*, vol. 22, no. 3–4, p. 237–254, 2006.
- Public site, mailing lists, SVN tree, examples, etc.:
<http://libmesh.sf.net/>

